Location: w32std.h
Link against: ws32.lib

Class RWsSession

RWsSession

Support

Supported from 5.0

Description

Window server session.

The session between the client and the window server can be used to mediate asynchronous events, and for client interface control and system control. A description of each of these capabilities is given below.

This class is not intended for user derivation.

Mediating asynchronous events:

Primarily, the session mediates asynchronous events to the user. Three event streams are available: the standard event stream which all applications must use; the redraw event stream which must be used by all applications except those which exclusively use backed-up windows; and the priority key event stream which may be used for abort keys and the like for specialist applications.

All these events are mediated as standard asynchronous services. Typical window server client programs encapsulate each service they require in an active object whose RunL() identifies the event and calls the appropriate member function of a class associated with an application framework, or a window.

Client interface control:

The client's interface with the window server has several aspects, each of which is controlled through the window server session.

- Flushing defines how requests to the window server are handled.

System control:

Many system-wide settings may be controlled by any application through its window server session. Typically, these functions are only used by the system shell and its associated sessions/applications.

- Auto-repeat and double-click.
- Query all window groups in the system.
- Setting the default shadow vector
- Setting the system pointer cursors
- Counting resources used by the window server. This is only useful for debugging checks.
- Getting and checking the state of the modifier keys (SHIFT, CTRL AND FN)
- Setting the window server background colour

Derivation

MWsClientClass	Base class for all classes whose objects are clients of the window server engine
RHandleBase	A handle to an object
RSessionBase	Client-side handle to a session with a server
RWsSession	Window server session

Defined in RWsSession:
ClaimSystemPointerCursorList(), ClearDefaultSystemPointerCursor(), ClearHotKeys(), ClearSystemPointerCursor(), Close(), ComputeMode(), Connect(), ELoggingDisable, ELoggingEnable, ELoggingHeapDump, ELoggingStatusDump, EPriorityControlComputeOff, EPriorityControlComputeOn, EPriorityControlDisabled, EventReady(), EventReadyCancel(), FindWindowGroupIdentifier(), FindWindowGroupIdentifier(), Flush(), FreeSystemPointerCursorList(), GetBackgroundColor(), GetColorModeList(), GetDefModeMaxNumColors(), GetDefaultOwningWindow(), GetDoubleClickSettings(), GetEvent(), GetFocusWindowGroup(), GetKeyboardRepeatRate(), GetModifierState(), GetPriorityKey(), GetRedraw(), GetWindowGroupClientThreadId(), GetWindowGroupHandle(), GetWindowGroupNameFromIdentifier(), GetWindowGroupOrdinalPriority(), HeapCount(), HeapSetFail(), LogCommand(), LogMessage(), NumWindowGroups(), NumWindowGroups(), PasswordEntered(), PointerCursorArea(), PointerCursorArea(), PointerCursorMode(), PointerCursorPosition(), PriorityKeyReady(), PriorityKeyReadyCancel(), PurgePointerEvents(), RWsSession(), RedrawReady(), RedrawReadyCancel(), RequestOffEvents(), ResourceCount(), RestoreDefaultHotKey(), SendEventToAllWindowGroups(), SendEventToAllWindowGroups(), SendEventToWindowGroup(), SendMessageToAllWindowGroups(), SendMessageToAllWindowGroups(), SendMessageToWindowGroup(), SendOffEventsToShellWindow(), SetAutoFlush(), SetBackgroundColor(), SetDefaultFadingParameters(), SetDefaultSystemPointerCursor(), SetDoubleClick(), SetHotKey(), SetKeyboardRepeatRate(), SetModifierState(), SetPointerCursorArea(), SetPointerCursorArea(), SetPointerCursorMode(), SetPointerCursorPosition(), SetRemoveKeyCode(), SetShadowVector(), SetSystemPointerCursor(), SetWindowGroupOrdinalPosition(), ShadowVector(), SimulateKeyEvent(), SimulateKeyEvent(), SimulateRawEvent(), TComputeMode, TLoggingCommand, Version(), WindowGroupList(), WindowGroupList()

Inherited from MWsClientClass:
WsHandle()

Inherited from RHandleBase:
Duplicate(), Handle(), SetHandle()

Inherited from RSessionBase:
Attach(), CreateSession(), EAutoAttach, EExplicitAttach, Send(), SendReceive(), SetRetry(), Share(), TAttachMode

---

Construction and destruction

RWsSession()

```
RWsSession();
```

Description

Default C++ constructor.

Constructs an uninitialised window server session. Note that it does not establish a connection to the window server — this must be done explicitly by calling the session's Connect() function. Before Connect() is called, no corresponding session object exists in the server, and the RWsSession contains no meaningful handle.

Connect()

```
TInt Connect();
```

Description

Connects the client session to the window server.

Connect() should be the first function called on an RWsSession object after it is created. The function establishes a connection to the window server, creating a corresponding session object in the server. Each session has one and only one connection to the server — attempting to call Connect() when a connection has already been made will cause a panic.

After a connection has been successfully established, all events are delivered to the client application through the RWsSession object.

Return value

TInt KErrNone if successful, otherwise another of the system-wide error codes.

Close()

```
void Close();
```

Description

Closes the window server session.

This function cleans up all resources in the `RWsSession` and disconnects it from the server. Prior to disconnecting from the window server the client-side window server buffer is destroyed without being flushed. This function should be called when the `RWsSession` is no longer needed — normally just before it is destroyed.

Version()

```
TVersion Version();
```

Description

Gets the window server version.

Return value

`TVersion`	Window server version containing major and minor version numbers, and build version.

Member functions

ClaimSystemPointerCursorList()

```
TInt ClaimSystemPointerCursorList();
```

Description

Claims the system pointer cursor list. A client must call this function before it can call `SetSystemPointerCursor()` or `ClearSystemPointerCursor()`.

Return value

`TInt`	`KErrNone` if successful, `KErrInUse` if another client is already using the system pointer cursor list, otherwise another of the system-wide error codes.

See also:

- `FreeSystemPointerCursorList()`

ClearDefaultSystemPointerCursor()

```
void ClearDefaultSystemPointerCursor();
```

Support

Supported from 6.0

Description

Clears the default system pointer cursor.

This sets the pointer to the current default system pointer cursor to NULL.

Panic codes

42	If this function is called by a client other than the owner of the system pointer cursor list.

ClearHotKeys()

```
TInt ClearHotKeys(THotKey aType);
```

Description

Clears all mappings for the specified hotkey, including the default mapping.

Hotkeys allow standard functions to be performed by application-defined key combinations.

Parameters

`THotKey aType` The hot key to be cleared

Return value

`TInt` `KErrNone` if successful, otherwise one of the system-wide error codes.

See also:

- SetHotKey()
- RestoreDefaultHotKey()

ClearSystemPointerCursor()

```
void ClearSystemPointerCursor(TInt aCursorNumber);
```

Description

Clears a system pointer cursor from the list.

Before calling this function, the client must first gain access to the list by calling `ClaimSystemPointerCursorList()`.

Parameters

`TInt aCursorNumber` Cursor number to clear

ComputeMode()

```
void ComputeMode(TComputeMode aMode);
```

Description

Sets the mode used to control process priorities.

The default window server behaviour is that the application that owns the window with keyboard focus gets foreground process priority (`EPriorityForeground`) while all other clients get background priority (`EPriorityBackground`). This function can be used to over-ride this default behaviour, as discussed in `TComputeMode`.

Note:

- Unlike real Symbian devices, the Emulator runs on a single process. As a result, on the Emulator this function sets the priority of individual threads rather than of processes. The values used for thread priorities are `EPriorityLess` instead of `EPriorityBackground`, and `EPriorityNormal` instead of `EPriorityForeground`.

Parameters

`TComputeMode aMode` The compute mode.

EventReady()

```
void EventReady(TRequestStatus* aStat);
```

Description

Requests standard events from the window server.

Standard events include all events except redraws and priority key events.

The client application will typically handle the completed request using the `RunL()` function of an active object, and in this case the request status `aStat` should be the `iStatus` member of that `CActive` object.

Note:

- The active object runs when an event is waiting — you should call `GetEvent()` in the `RunL()` function to get the event.
- You should not call this function again until you've either called `GetEvent()` or `EventReadyCancel()`

Parameters

`TRequestStatus* aStat`	Request status. On successful completion contains `KErrNone`, otherwise another of the system-wide error codes.

See also:

- CActive
- GetEvent()

EventReadyCancel()

```
void EventReadyCancel();
```

Description

Cancels a request for standard events from the window server. This request was made using `EventReady()`.

The client application will typically use an active object to handle standard events, and this function should be called from the active object's `DoCancel()` function.

See also:

- EventReady()

FindWindowGroupIdentifier()

```
TInt FindWindowGroupIdentifier(TInt aPreviousIdentifier,const TDesC& aMatch,TInt aOffset=0);
```

Description

Gets all window groups whose names match a given string — which can contain wildcards.

An example use of this function might be in order to find all the currently running copies of a particular application, assuming that the window group name contained the application name. An optional argument, `aOffset`, specifies the number of characters to be ignored at the beginning of the window group name. As several window group names may match the given string, and the function can return only one at a time, there is an argument, `aPreviousIdentifier`, which gives the identifier for the previous match that was returned. In other words, it means, "get me the next match after this one." The first time the function is called, give 0 as the previous identifier.

Matching is done using `TDesC::MatchF()`, which does a folded match. Wildcards '*' and '?' can be used to denote one or more characters and exactly one character, respectively. Windows are searched in front to back order.

Parameters

`TInt aPreviousIdentifier`	Identifier of the last window group returned. If the value passed is not a valid identifier, the function returns `KErrNotFound`.
`const TDesC& aMatch`	String to match window group name against: may contain wildcards
`TInt aOffset=0`	The first `aOffset` characters of the window group name are ignored when doing the match.

Return value

`TInt`	The next window group, after the one identified by `aPreviousIdentifier`, whose name matches `aMatch`. `KErrNotFound` if no window group matched or `aPreviousIdentifier` was not a valid identifier.

FindWindowGroupIdentifier()

```
TInt FindWindowGroupIdentifier(TInt aPreviousIdentifier, TThreadId aThreadId);
```

Description

Gets the identifiers of window groups belonging to a client which is owned by a thread with the specified thread ID.

The thread may own more than one window group, so the identifier returned is the one after `aPreviousIdentifier`. The first time the function is called, use 0 for the previous identifier.

Parameters

`TInt aPreviousIdentifier`	Identifier returned by previous call
`TThreadId aThreadId`	Thread owning the window group or groups.

Return value

`TInt`	Next window group identifier after the one identified by `aPreviousIdentifier`

Flush()

```
void Flush();
```

Description

Sends all pending commands in the buffer to the window server.

Delivering a command to the window server requires a context switch, and so it is more efficient to deliver several commands in one go. Hence all client commands are normally first placed into a buffer for delivery to the window server.

The buffer is delivered when:

- It gets full.
- A command that returns a value is called (there are a few exceptions to this).
- This function is called.

Note:

- This function is called when a prompt response is required from the window server, e.g. after doing some drawing.

FreeSystemPointerCursorList()

```
void FreeSystemPointerCursorList();
```

Description

Releases the system pointer cursor list and deletes all the entries in it.

A client should call this function when it no longer needs the system pointer cursor list.

GetBackgroundColor()

```
TRgb GetBackgroundColor() const;
```

Description

Gets the window server background colour.

Return value

`TRgb`	Background colour

GetColorModeList()

```
TInt GetColorModeList(CArrayFixFlat<TInt> *aModeList) const;
```

Description

Gets the list of available colour modes.

Note:

- This function should usually be called within `User::LeaveIfError()`.
- The only time that an error can be generated is when the array gets resized to the number of display modes. Thus if you make the size of your array equal to the maximum number of display modes over all hardware then this function will never leave. Currently there can only ever be 11 as that is the number of different values in `TDisplayMode`.
- This function was not `const` prior to version 6.0.

Parameters

`CArrayFixFlat<TInt> * aModeList`	On return, contains a `TDisplayMode` entry for each of the available display modes. Note that the array's contents are deleted prior to filling with display mode information.

Return value

`TInt`	`KErrNone` if successful. `KErrNoMemory` if `aModeList` must be re-sized to hold all the display modes and this results in an OOM error.

GetDefaultOwningWindow()

```
TInt GetDefaultOwningWindow();
```

Description

Gets the identifier of the current default owning window group.

Return value

`TInt`	Identifier of current default owning window group. Returns 0 if there isn't one.

GetDefModeMaxNumColors()

```
TDisplayMode GetDefModeMaxNumColors(TInt& aColor,TInt& aGray) const;
```

Description

Gets the number of colours available in richest supported colour mode, the number of greys available in the richest grey mode, and returns the default display mode.

Note:

- This function was not `const` prior to version 6.0.

Parameters

`TInt& aColor`	The number of colours in the richest supported colour mode.
`TInt& aGray`	The number of greys in the richest supported grey mode.

Return value

`TDisplayMode`	The default display mode.

GetDoubleClickSettings()

```
void GetDoubleClickSettings(TTimeIntervalMicroSeconds32 &aInterval, TInt &aDistance);
```

Description

Gets the current system-wide settings for pointer double clicks.

Note:

- Double click distances are measured in pixels as the sum of the X distance moved and the Y distance moved between clicks. For example: a first click at 10, 20 and a second click at 13,19 gives a distance of (13-10)+(21-20) = 4.

Parameters

TTimeIntervalMicroSeconds32& aInterval	Maximum interval between clicks that constitutes a double click
TInt &aDistance	Maximum distance between clicks that constitutes a double click

See also:

- SetDoubleClick()

GetEvent()

```
void GetEvent(TWsEvent& aEvent);
```

Description

Gets a standard event from the session for processing.

The type of event returned by GetEvent() may be any of those listed in TEventCode. To access the data within an event, the event should be converted to the appropriate type, using functions provided by the TWsEvent class — TWsEvent also provides a function to find out the type of the event.

Note:

- It is possible that the returned event is of type EEventNull. Clients should normally ignore these events.
- This function should only be called in response to notification that an event has occurred — otherwise the client will be panicked.

  This function would normally be called in the RunL() function of an active object which completes with the EventReady() function's request status.

Parameters

TWsEvent& aEvent	On return, contains the event that occurred

See also:

- TEventCode
- EventReady()

GetFocusWindowGroup()

```
TInt GetFocusWindowGroup();
```

Description

Gets the identifier of the window group that currently has the keyboard focus.

Note:

- This might not necessarily be the front-most window group, as window groups can disabled keyboard focus.

Return value

TInt	Identifier of window group with keyboard focus.

GetKeyboardRepeatRate()

```
void GetKeyboardRepeatRate(TTimeIntervalMicroSeconds32& aInitialTime, TTimeIntervalMicroSeconds32& aTime);
```

Description

Gets the current system-wide settings for the keyboard repeat rate.

Parameters

TTimeIntervalMicroSeconds32& aInitialTime	Time before first repeat key event
TTimeIntervalMicroSeconds32& aTime	Time between subsequent repeat key events

See also:

- SetKeyboardRepeatRate()

GetModifierState()

```
TInt GetModifierState() const;
```

Description

Get the state of modifier keys.

The state of each modifier key (defined in TEventModifier) is returned in a bitmask.

The modifier keys are: EModifierLeftAlt, EModifierRightAlt, EModifierAlt, EModifierLeftCtrl, EModifierRightCtrl, EModifierCtrl, EModifierLeftShift, EModifierRightShift, EModifierShift, EModifierLeftFunc, EModifierRightFunc, EModifierFunc, EModifierCapsLock, EModifierNumLock, EModifierScrollLock, , , ,

Return value

TInt	Modifier state

See also:

- SetModifierState()

GetPriorityKey()

```
void GetPriorityKey(TWsPriorityKeyEvent& aEvent);
```

Description

Gets the completed priority key event from the window server session.

Priority key events are typically used for providing "Abort" or "Escape" keys for an application.

This function is similar to GetEvent(), except that it returns a TWsPriorityKeyEvent instead of a TWsEvent.

Note:

- This should only be called after notification that a priority key event has occurred.

Parameters

TWsPriorityKeyEvent& aEvent	On return, contains the priority key event that occurred.

See also:

- PriorityKeyReady()
- PriorityKeyReadyCancel()

GetRedraw()

```
void GetRedraw(TWsRedrawEvent& aEvent);
```

Description

Gets the redraw event from the session.

This function is similar to GetEvent(), except that the event is returned as a TWsRedrawEvent, and hence there is no need to convert it from a TWsEvent.

The function should only be called after notification that a redraw is waiting.

Parameters

`TWsRedrawEvent& aEvent`	On return, contains the redraw event that occurred

See also:

- `RedrawReady()`
- `GetEvent()`
- `CActive`

GetWindowGroupClientThreadId()

```
TInt GetWindowGroupClientThreadId(TInt aIdentifier, TThreadId &aThreadId);
```

Description

Gets the thread ID of the client that owns the window group specified by the window group identifier.

Parameters

`TInt aIdentifier`	The window group identifier
`TThreadId &aThreadId`	On return, contains the thread ID

Return value

`TInt`	`KErrNone` if successful, otherwise another of the system-wide error codes.

GetWindowGroupHandle()

```
TInt GetWindowGroupHandle(TInt aIdentifier);
```

Description

Gets the handle of the window specified by the window group identifier.

This is the handle that was passed as an argument to `RWindowGroup::Construct()`.

Parameters

`TInt aIdentifier`	The window group identifier

Return value

`TInt`	Handle of the window group

GetWindowGroupNameFromIdentifier()

```
TInt GetWindowGroupNameFromIdentifier(TInt aIdentifier, TDes& aWindowName);
```

Description

Gets the name of a window group from its identifier.

Using the list of identifiers returned by `WindowGroupList()`, it is possible to get the names of all window groups in the system. Note that window names are a zero length string by default.

Note:

- Note that the window group name must have been previously set using `RWindowGroup::SetName()` to contain a meaningful value.

Parameters

`TInt aIdentifier`	The identifier of the window group whose name is to be inquired
`TDes& aWindowName`	On return, contains the window group name

Return value

TInt	KErrNone if successful, otherwise another of the system-wide error codes.

GetWindowGroupOrdinalPriority()

```
TInt GetWindowGroupOrdinalPriority(TInt aIdentifier);
```

Description

Gets a window group's priority.

Parameters

TInt aIdentifier	The window group identifier

Return value

TInt	The window group priority

HeapCount()

```
TInt HeapCount() const;
```

Description

Gets the heap count.

This function calls RHeap::Count() on the window server's heap, after throwing away all the temporary objects allocated for each window.

Return value

TInt	The window server's heap count.

HeapSetFail()

```
void HeapSetFail(RHeap::TAllocFail aType,TInt aValue);
```

Description

Sets the heap failure mode in the window server.

The release version of the base does not support simulated heap failure functionality, and the result of this function is additional error messages. In the debug version the clients are notified of the simulated failure and handle it.
See RHeap::__DbgSetAllocFail() for more information.

Note:

- It is unlikely, but possible to create a ROM with a mixture of Release and Debug versions of the Base and Window Server DLLs, which results in different behaviour to that described above. If you run a debug Window Server with a release version of the Base, then calling this function will result in neither extra error messages (e.g. EDrawingRegion) nor simulated heap failures. However if you have a release Window Server with a debug Base then you will get both simulated heap failures and the extra error messages.

Parameters

RHeap::TAllocFail aType	An enumeration which indicates how to simulate heap allocation failure.
TInt aValue	The rate of failure; when aType is RHeap::EDeterministic, heap allocation fails every aRate attempt

See also:

- RHeap::__DbgSetAllocFail()

LogCommand()

```
void LogCommand(TLoggingCommand aCommand);
```

Support

Supported from 7.0

Description

Allows the window server client to enable or disable logging of window server events.

The type of logging that takes place (e.g. whether to file or to serial port) depends on the settings specified in the wsini.ini file.

Clients can also force a dump of the window tree or information about the window server's heap size and usage.

Parameters

`TLoggingCommand aCommand` The logging command.

Notes:

- For logging to work, the wsini.ini file has to specify the type of logging required and the DLLs for that type of logging must have been correctly installed. Otherwise, calling this function will have no effect.

See also:

- wsini.ini

LogMessage()

```
void LogMessage(const TLogMessageText &aMessage);
```

Description

Adds a message to the window server debug log if one is currently in operation.

Parameters

`const TLogMessageText& aMessage` The text added to the message log.

NumWindowGroups()

```
TInt NumWindowGroups() const;
```

Description

Gets the total number of window groups currently running in the window server.

This includes all the groups running in all sessions.

Return value

`TInt` Total number of window groups running in the server

NumWindowGroups()

```
TInt NumWindowGroups(TInt aPriority) const;
```

Description

Gets the number of window groups of a given window group priority running in all sessions in the window server.

Parameters

`TInt aPriority` Window group priority

Return value

TInt Number of window groups of priority aPriority

PasswordEntered()

```
void PasswordEntered();
```

Description

Disables window server password mode.

This function must be called by the session which owns the password window when the correct machine password has been entered.

PointerCursorArea()

```
TRect PointerCursorArea() const;
```

Support

Supported from 6.0

Description

Gets the pointer cursor area for the first screen display mode.

This is the area of the screen in which the virtual cursor can be used while in relative mouse mode. While in pen or mouse mode the event co-ordinates are forced to be within this area unless you click outside the drawable area.

Return value

TRect The pointer cursor area for the first screen display mode.

See also:

- SetPointerCursorArea()

PointerCursorArea()

```
TRect PointerCursorArea(TInt aScreenSizeMode) const;
```

Support

Supported from 6.0

Description

Gets the pointer cursor area for the specified screen display mode.

This is the area of the screen in which the virtual cursor can be used while in relative mouse mode. While in pen or mouse mode the event co-ordinates are forced to be within this area unless you click outside the drawable area.

Parameters

TInt aScreenSizeMode The screen mode for which the pointer cursor area is required.

Return value

TRect The pointer cursor area for the specified screen display mode.

See also:

- SetPointerCursorArea()

PointerCursorMode()

```
TPointerCursorMode PointerCursorMode() const;
```

Support

Supported from 6.0

Description

Gets the current mode for the pointer cursor.

The mode determines which sprite is used for the pointer cursor at any point.

Return value

`TPointerCursorMode`	The current mode for the pointer cursor.

PointerCursorPosition()

```
TPoint PointerCursorPosition() const;
```

Support

Supported from 6.0

Description

Gets the pointer cursor position.

This function allows an application to determine the position of the virtual cursor.

Return value

`TPoint`	The position of the virtual cursor.

PurgePointerEvents()

```
void PurgePointerEvents();
```

Description

Removes all pointer events waiting to be delivered to this session.

The events are removed from the event queue without being processed. This might occur, for example, at application startup.

PriorityKeyReady()

```
void PriorityKeyReady(TRequestStatus *aStat);
```

Description

Requests priority key events from the window server.

Typically, an client will create an active object for priority key events with a higher priority than the active objects for standard events. The client will then normally handle completed priority key requests in the active object's `RunL()` function.

As in `EventReady()`, the request status argument should be the set to the `iStatus` member of `CActive`. When priority key events occur, they are obtained using `GetPriorityKey()`.

Note:

- You should not call this function again until you've either called `GetPriorityKey()` or `PriorityKeyReadyCancel()`.

Parameters

`TRequestStatus* aStat`	Request status. On successful completion contains `KErrNone`, otherwise another of the system-wide error codes.

See also:

- PriorityKeyReadyCancel()

- GetPriorityKey()
- CActive

PriorityKeyReadyCancel()

```
void PriorityKeyReadyCancel();
```

Description

Cancels a priority key event request.

If active objects are used, this function should be called from the active object's `DoCancel()` function.

See also:

- PriorityKeyReady()
- CActive

RedrawReady()

```
void RedrawReady(TRequestStatus* aStat);
```

Description

Requests redraw events from the window server.

Typically, a client will create an active object for redraw events with a lower priority than the active objects for standard events. The client will then typically handle completed redraw requests in the active object's `RunL()` function.

As in `EventReady()`, the request status `aStat` should be used as the `iStatus` member of an active object. When a redraw event occurs the active object's `RunL()` function is called — the redraw event can be obtained by calling `GetRedraw()` in the `RunL()`.

Note:

- You should not call this function again until you've either called `GetRedraw()` or `RedrawReadyCancel()`.

Parameters

`TRequestStatus* aStat`	The request status. On successful completion contains `KErrNone`, otherwise another of the system-wide error codes.

See also:

- RedrawReadyCancel()
- GetRedraw()
- CActive

RedrawReadyCancel()

```
void RedrawReadyCancel();
```

Description

Cancels a redraw event request.

If active objects are used, this function should be called from the active object's `DoCancel()` function.

See also:

- RedrawReady()

RequestOffEvents()

```
TInt RequestOffEvents(TBool aOn,RWindowTreeNode* aWin=NULL);
```

Support

Supported from 6.0

Description

Requests the window server to send OFF events to a window.

This replaces `SendOffEventsToShellWindow()` in versions prior to 6.0.

This function is called by a client to command the window server to send it OFF events. The window server then sends the window OFF events when an action occurs which requires power down, rather than handling powering down itself.

Notes:

- Any client can ask for OFF events, but only one window in the system can be set to receive them. If this function is called when another window is set to receive OFF events then the client will be panicked. The exception is the shell, which is allowed to take receipt of OFF events from other clients.
- The window server identifies the shell client by comparing the process name of the client with the process name of the shell. Only the first client created by the shell is guaranteed to have the extra shell client privileges.
- Prior to ER4, the window server handled all powering down. However since powering down is policy dependent it can be handled by the shell.
- This function does not automatically flush the client side buffer, and hence does not become active until the buffer is flushed. The shell should arrange for a function that does flush the buffer to be called soon afterwards. See `Flush()`.
- If the shell dies or terminates just before the action requiring power down happens then the window server will handle it rather than passing it on to the shell.
- The window server has a queue of messages that it is waiting to send to clients. If the shell's client's queue is full and the window server cannot make room for the OFF message then it will power down the machine itself.

Parameters

`TBool aOn`	`ETrue` to get the window server to send `EEventShellSwitchOff` messages to the shell (rather than powering down). `EFalse` makes the window server switch back to powering down the machine itself.
`RWindowTreeNode * aWin=NULL`	The handle to the window or window group of the shell — to which the message is to be sent. This may be NULL only if `aOn=EFalse`, i.e. the window server is handling power down itself. If `aOn=ETrue` then this must not be NULL, or the Window Server will panic the shell. Note that as far as the window server is concerned the handle must exist when this function is called.

Return value

`TInt`	`KErrNone` if successful, otherwise another of the system-wide error codes.

Panic codes

51	If this function is called when a window has already been set to receive OFF events. The exception is the shell, which may take the receipt of OFF events away from other windows.

ResourceCount()

```
TInt ResourceCount();
```

Description

Gets the number of objects that the server has allocated for that client.

This function can be used to check that the client has correctly cleaned up all of its objects.

Return value

`TInt`	The number of resources allocated to the client.

RestoreDefaultHotKey()

```
TInt RestoreDefaultHotKey(THotKey aType);
```

Description

Restores the default mapping for a hot key.

The function clears current mappings for a hot key and restores the default mapping — see `THotKey` for the default.

Parameters

`THotKey aType`	The hot key to restore to its default value

Return value

`TInt`	`KErrNone` if successful, otherwise another of the system-wide error codes.

See also:

- ClearHotKeys()

SendEventToAllWindowGroups()

```
TInt SendEventToAllWindowGroups(const TWsEvent& aEvent);
```

Support

Supported from 5.1

Description

Sends the specified event to all existing window groups.

Parameters

`const TWsEvent& aEvent`	The event to be sent to all window groups.

Return value

`TInt`	`KErrNone` if successful, otherwise another of the system-wide error codes.

SendEventToAllWindowGroups()

```
TInt SendEventToAllWindowGroups(TInt aPriority, const TWsEvent& aEvent);
```

Support

Supported from 5.1

Description

Sends the specified event to all window groups with the specified priority.

Parameters

`TInt aPriority`	The priority which window groups must have to be sent the message.
`const TWsEvent& aEvent`	The event to be sent to the specified window groups.

Return value

`TInt`	`KErrNone` if successful, otherwise another of the system-wide error codes.

SendEventToWindowGroup()

```
TInt SendEventToWindowGroup(TInt aIdentifier, const TWsEvent& aEvent);
```

Description

Sends an event to a window group.

Parameters

TInt aIdentifier	Window group identifier
const TWsEvent& aEvent	Event to send to window group

Return value

TInt	KErrNone if successful, otherwise another of the system-wide error codes.

SendMessageToAllWindowGroups()

```
TInt SendMessageToAllWindowGroups(TUid aUid, const TDesC8& aParams);
```

Support

Supported from 6.0

Description

Sends a message to all window groups.

Note:

- In order to receive messages sent using this function you will need to implement the MCoeMessageObserver interface which is defined in the UI Control Framework API.

Parameters

TUid aUid	A UID which uniquely identifies the session sending the message.
const TDesC8& aParams	The message data. An unlimited amount of data can be passed in this argument.

Return value

TInt	KErrNone if successful, otherwise another of the system-wide error codes.

SendMessageToAllWindowGroups()

```
TInt SendMessageToAllWindowGroups(TInt aPriority, TUid aUid, const TDesC8& aParams);
```

Support

Supported from 6.0

Description

Sends a message to all window groups with the specified priority.

Note:

- In order to receive messages sent using this function you will need to implement the MCoeMessageObserver interface which is defined in the UI Control Framework API.

Parameters

TInt aPriority	The priority which window groups must have to be sent the message.
TUid aUid	A UID which uniquely identifies the session sending the message.
const TDesC8& aParams	The message data. An unlimited amount of data can be passed in this argument.

Return value

TInt	KErrNone if successful, otherwise another of the system-wide error codes.

SendMessageToWindowGroup()

```
TInt SendMessageToWindowGroup(TInt aIdentifier, TUid aUid, const TDesC8& aParams);
```

Description

Sends a message to a window group.

The window group will then receive an event of type `EEventMessageReady` notifying it that a message has been received. The window group can belong to this or another session.

Pre version 6.0:

Prior to version 6.0 the message can then be retrieved using `RWindowGroup::FetchMessage()`.

An application must ensure that it calls `RWindowGroup::FetchMessage()` whenever it receives an `EEventMessageReady` event. It will receive no further `EEventMessageReady` events until it does so. However application developers using the standard GUI framework should note that `CEikAppUi` calls `RWindowGroup::FetchMessage()` automatically.

Version 6.0 and onward:

From version 6.0 `RWindowGroup::FetchMessage()` has been removed. In order to receive messages sent using this function you will need to implement the `MCoeMessageObserver` interface which is defined in the UI Control Framework API.

Parameters

`TInt aIdentifier`	The identifier of the window group to receive the message.
`TUid aUid`	A UID which uniquely identifies the session sending the message.
`const TDesC8& aParams`	The message data. An unlimited amount of data can be passed in this argument.

Return value

`TInt` `KErrNone` if successful, otherwise another of the system-wide error codes.

See also:

- MCoeMessageObserver

SendOffEventsToShellWindow()

```
void SendOffEventsToShellWindow(TBool aOn,RWindowTreeNode *aWin=NULL);
```

Support

Withdrawn in 6.0

Description

Requests the window server to send OFF events to the shell window.

This function is called by the shell to command the window server to send it OFF events. The window server then sends the shell OFF events when an action occurs which requires power down, rather than handling powering down itself.

Notes:

- Prior to ER4, the window server handled all powering down. However since powering down is policy dependent it can be handled by the shell.
- The window server identifies the shell client by comparing the process name of the client with the process name of the shell. Only the first client created by the shell is guaranteed to have the extra shell client privileges.
- This function does not automatically flush the client side buffer, and hence does not become active until the buffer is flushed. The shell should arrange for a function that does flush the buffer to be called soon afterwards. See `Flush()`.
- If the shell dies or terminates just before the action requiring power down happens then the window server will handle it rather than passing it on to the shell.
- The window server has a queue of messages that it is waiting to send to clients. If the shell's client's queue is full and the window server cannot make room for the OFF message then it will power down the machine itself.
- In version 6.0 this function is replaced by:

  ```
  TInt RequestOffEvents(TBool aOn,RWindowTreeNode *aWin=NULL);
  ```

Parameters

TBool aOn	ETrue to get the window server to send EEventShellSwitchOff messages to the shell (rather than powering down). EFalse makes the window server switch back to powering down the machine itself.
RWindowTreeNode * aWin=NULL	The handle to the window or window group of the shell — to which the message is to be sent. This may be NULL only if aOn=EFalse, i.e. the window server is handling power down itself. If aOn=ETrue then this must not be NULL, or the Window Server will panic the shell. Note that as far as the window server is concerned the handle must exist when this function is called.

SetAutoFlush()

```
TBool SetAutoFlush(TBool aState);
```

Description

Sets a session's auto-flush state.

If auto-flush is set to ETrue, the window server buffer is flushed immediately anything is put into it, instead of waiting until it becomes full. This setting is normally used only in a debugging environment.

If the auto-flush state is EFalse, the window server buffer is flushed normally.

Parameters

TBool aState	ETrue to set auto-flushing on, EFalse to disable auto-flushing.

Return value

TBool	Previous auto-flush state

SetBackgroundColor()

```
void SetBackgroundColor(TRgb aColor);
```

Description

Sets the background colour for the window server.

This background can only be seen in areas of the display that have no windows on them: so for many applications it will never be seen — it affects no other windows.

Parameters

TRgb aColor	Background colour

SetDefaultFadingParameters()

```
void SetDefaultFadingParameters(TUint8 aBlackMap,TUint8 aWhiteMap);
```

Support

Supported from 6.0

Description

Sets the default fading parameters.

Fading is used to change the colour of a window to make other windows stand out. For example, you would fade all other windows when displaying a dialogue. This function sets whether, and the amount by which, fading makes a faded window closer to white or closer to black.

Fading re-maps colours in the faded window to fall between the specified black and white map values. If aBlackMap=0 and aWhiteMap=255 then the colours are mapped unchanged. As the values converge the colours are mapped to a smaller range — so the differences between colours in the faded window decrease. If the values are reversed then the colours are inverted (i.e. where the unfade window would be black, it is now white).

Changing the default will automatically apply to current graphic contexts but will not have any affect on windows that are already faded.

Note:

- `RWindowTreeNode::SetFaded()` and `CWindowGc::SetFaded()` use these fading parameters, and allow control over the exact colour mapping used when fading. In addition, the `RWindowTreeNode` functions have variants which allow these default parameters to be over-ridden when they are called.

Parameters

`TUint8 aBlackMap`	Black map fading parameter.
`TUint8 aWhiteMap`	White map fading parameter.

See also:

- RWindowTreeNode::SetFaded()
- CWindowGc::SetFadingParameters()

SetDefaultSystemPointerCursor()

```
void SetDefaultSystemPointerCursor(TInt aCursorNumber);
```

Support

Supported from 6.0

Description

Sets the default system pointer cursor.

This function can only be called by the owner of the system pointer cursor list. By default the 0th entry in the pointer cursor list is assigned as the system pointer. The function allows any cursor from the list or even no cursor to be set as the system pointer cursor.

Notes

- Ownership of the system pointer cursor list can be obtained by calling `ClaimSystemPointerCursorList()` when no-one else has ownership.

Parameters

`TInt aCursorNumber`	The index of the new default system pointer cursor within the system cursor list.

Panic codes

If this function is called when this session is not the owner of the system pointer cursor list.

SetDoubleClick()

```
void SetDoubleClick(const TTimeIntervalMicroSeconds32 &aInterval, TInt aDistance);
```

Description

Sets the system-wide double click settings.

Note:

- Double click distance is measured, in pixels, as the sum of the X distance moved and the Y distance moved between clicks. For example: a first click at 10, 20 and a second click at 13,19 gives a distance of (13-10)+(21-20) = 4.

Parameters

`const TTimeIntervalMicroSeconds32& aInterval`	Maximum interval between clicks that constitutes a double click
`TInt aDistance`	Maximum distance between clicks that constitutes a double click

See also:

- GetDoubleClickSettings()

SetHotKey()

```
TInt SetHotKey(THotKey aType, TUint aKeyCode, TUint aModifierMask, TUint aModifier);
```

Description

Sets hot keys.

Hotkeys allow standard functions to be performed by application-defined key combinations.

This function maps any key press (with optional modifiers) to one of the hot keys defined in `THotKey`. More than one key combination may be mapped to each hot key: a new mapping is added each time the function is called.

Modifier key states are defined in `TEventModifier`. The modifiers that you want to be in a particular state should be specified in `aModifierMask` and the ones of these you want to be set should be specified in `aModifiers`. For example, if you want to capture FN-A and you want the SHIFT modifier unset, but you don't care about the state of the other modifiers then set both the flags for SHIFT and FN in `aModiferMask` and only set FN in `aModifiers`.

Note:

- Default hotkey settings exist, but this function can be used for customisation. Typically it might be be used by a shell application or other application that controls system-wide settings.

Parameters

`THotKey aType`	The hot key to be mapped
`TUint aKeyCode`	The keycode to be mapped to the hot key
`TUint aModifierMask`	Modifier keys to test for a match
`TUint aModifier`	Modifier keys to be tested for "on" state

Return value

`TInt` `KErrNone` value if successful, otherwise another of the system-wide error codes.

SetKeyboardRepeatRate()

```
void SetKeyboardRepeatRate(const TTimeIntervalMicroSeconds32& aInitialTime, const TTimeIntervalMicroSeconds32& aTime);
```

Description

Sets the system-wide keyboard repeat rate.

This is the rate at which keyboard events are generated when a key is held down.

Note:

- The default settings for the keyboard repeat rate are 0.3 seconds for the initial delay, and 0.1 seconds for the interval between subsequent repeats. However, since the settings are system-wide, these will not necessarily be the current settings when an application is launched: the settings may have been over-ridden by another module.

Parameters

`const TTimeIntervalMicroSeconds32& aInitialTime`	Time before first repeat key event
`const TTimeIntervalMicroSeconds32& aTime`	Time between subsequent repeat key events

See also:

- GetKeyboardRepeatRate()

SetModifierState()

```
void SetModifierState(TEventModifier aModifier,TModifierState aState);
```

Description

Sets the state of the modifier keys.

This function is typically used for permanent modifier states such as Caps Lock or Num Lock, but other possible uses include on-screen function key simulation, or the implementation of a Shift Lock key.

Parameters

`TEventModifier aModifier`	Modifier to set.
`TModifierState aState`	Modifier state.

See also:

- GetModifierState()

SetPointerCursorArea()

```
void SetPointerCursorArea(const TRect& aArea);
```

Support

Supported from 6.0

Description

Sets the area of the screen in which the virtual cursor can be used while in relative mouse mode, for the first screen display mode.

This function sets the area for the first screen mode — the one with index 0 — which in most devices will be the only screen mode. The following function overload can be used to set the screen area for other modes. The area is set and stored independently on each screen mode, so that it is not necessary to call this function again when switching back to the first screen mode.

The default area is the full digitiser area. When you set the area it will come into immediate affect, i.e., if necessary the current pointer position will be updated to be within the new area.

Notes:

- Relative mouse mode is where the events received from the base by window server are deltas from the last position of the pointer, as opposed to absolute co-ordinates.
- This function is honoured even if there is a mouse or pen (e.g. on the emulator), by mapping the co-ordinates of where you click into the area set using this function. However the function does not restrict clicks outside of the 'drawing area' on the Emulator, to allow you to select items on the fascia.

Parameters

`const TRect& aArea`	The area of the screen in which the virtual cursor can be used.

SetPointerCursorArea()

```
void SetPointerCursorArea(TInt aScreenSizeMode,const TRect& aArea);
```

Support

Supported from 6.0

Description

Sets the area of the screen in which the virtual cursor can be used while in relative mouse mode, for a specified screen display mode.

The default area is the full digitiser area for the given mode. When you set the area it will come into immediate affect, i.e., if necessary the current pointer position will be updated to be within the new area.

The area is set and stored independently on each screen mode, so that it is not necessary to call this function again when switching back to a mode.

Notes:

- Relative mouse mode is where the events received from the base by window server are deltas from the last position of the pointer, as opposed to absolute co-ordinates.
- The previous function overload may be used to set the screen area for only the first mode.
- This function is honoured even if there is a mouse or pen (e.g. on the emulator), by mapping the co-ordinates of where you click into the area set using this function. However the function does not restrict clicks outside of the 'drawing area' on the Emulator, to allow you to select items on the fascia.

Parameters

`TInt aScreenSizeMode`	The screen mode to which the new area applies.
`const TRect& aArea`	The area of the screen, in the `aScreenSizeMode` screen mode, within which the virtual cursor can be used.

SetPointerCursorMode()

```
void SetPointerCursorMode(TPointerCursorMode aMode);
```

Support

Supported from 6.0

Description

Sets the current mode for the pointer cursor.

The mode determines which sprite is used for the pointer cursor at any point. See `TPointerCursorMode` for more information.

Parameters

`TPointerCursorMode aMode`	The new mode for the pointer cursor.

See also:

- TPointerCursorMode

SetPointerCursorPosition()

```
void SetPointerCursorPosition(const TPoint& aPosition);
```

Support

Supported from 6.0

Description

Sets the pointer cursor position.

This function allows an application to move the virtual cursor. It works in all modes — not just relative mouse mode.

Note:

- The function works in screen co-ordinates and honours the pointer cursor area exactly as pen presses do — i.e. only when they are in the drawing area on the Emulator.

Parameters

`const TPoint& aPosition`	The new pointer cursor position.

SetRemoveKeyCode()

```
void SetRemoveKeyCode(TBool aRemove);
```

Support

Supported from 5.1

Description

Sets whether to remove the top 16 bits of keypress code — Emulator builds only.

This function allows the Emulator to use Windows to translate keypresses into the correct key code for each locale, rather than having to do the translation for each international keyboard itself.

The top 16 bits of a keypress code contains the keycode that Windows would produce if the key had been pressed in a typical Windows program. If `aRemove` is `EFalse` the client can get these top 16 bits — if the value is non-zero the `CKeyTranslator` uses it rather than calculating it's own value. If `aRemove` is `ETrue` the window server strips the top 16 bits of the scan code before passing the value on to the client.

Note:

- This function is intended for JAVA but it will be useful to any client who creates their own `CKeyTranslator` and processes keyups and downs.

Parameters

`TBool aRemove`	`ETrue` to remove the code (default), `EFalse` to pass it to the client.

SetShadowVector()

```
void SetShadowVector(const TPoint& aVector);
```

Description

Sets the shadow vector.

Parameters

`const TPoint& aVector`	New shadow vector

SetSystemPointerCursor()

```
TInt SetSystemPointerCursor(const RWsPointerCursor& aPointerCursor,TInt aCursorNumber);
```

Description

Sets a cursor in the system pointer cursor list.

To gain access to the list, the client must first call `ClaimSystemPointerCursorList()`.

Parameters

`const RWsPointerCursor& aPointerCursor`	Pointer cursor to set in the list
`TInt aCursorNumber`	Cursor number in the list

Return value

`TInt`	`KErrNone` if successful, otherwise another of the system-wide error codes.

SetWindowGroupOrdinalPosition()

```
TInt SetWindowGroupOrdinalPosition(TInt aIdentifier, TInt aPosition);
```

Description

Sets the ordinal position of a window group.

This function allows the caller to change the ordinal position of an existing window group. It would typically be used by a shell application.

Parameters

`TInt aIdentifier`	The window group
`TInt aPosition`	Ordinal position for the window group

Return value

`TInt`	`KErrNone` if successful, otherwise another of the system-wide error codes.

ShadowVector()

```
TPoint ShadowVector() const;
```

Description

Gets the current value of the shadow vector.

Return value

`TPoint` Current shadow vector

SimulateKeyEvent()

```
void SimulateKeyEvent(TRawEvent aEvent);
```

Support

Withdrawn in 6.0
Supported from 5.1

Description

Sends a simulated raw key event to the window server.

The function is only used for testing purposes.

Note:

- This function differs from `RWsSession::SimulateRawEvent()` in that the keys will not be processed by the translator table thus allowing you more control over the key that gets through. It is particular useful for sending key presses that change or toggle the backlight or contrast.
- The function is replaced by an overload with the same name, but taking a `TKeyEvent`, in version 6.0

Parameters

`TRawEvent aEvent` The raw key event to be sent.

SimulateKeyEvent()

```
void SimulateKeyEvent(TKeyEvent aEvent);
```

Support

Supported from 6.0

Description

Sends a simulated key event to the window server.

The function is only used for testing purposes.

Note:

- All the members of `TKeyEvent` are honoured except the `iRepeat` which is ignored and set to 0 by the key handling code.
- This function replaces an overload taking a `TRawEvent`.

Parameters

`TKeyEvent aEvent` The key event to be sent.

SimulateRawEvent()

```
void SimulateRawEvent(TRawEvent aEvent);
```

Description

Simulates raw events as though they come from the kernel.

Currently this function is used only for testing purposes. It is possible that it may in future be used by applications.

Parameters

`TRawEvent aEvent`	The raw event.

WindowGroupList()

```
TInt WindowGroupList(CArrayFixFlat<TInt>* aWindowList);
```

Description

Gets a list of identifiers of all window groups in all window server sessions.

An array buffer must be created to store the resultant list.

Parameters

`CArrayFixFlat<TInt>* aWindowList`	List of identifiers of all window groups in server

Return value

`TInt`	`KErrNone` if successful, otherwise another of the system-wide error codes.

WindowGroupList()

```
TInt WindowGroupList(TInt aPriority, CArrayFixFlat<TInt> *aWindowList);
```

Description

Lists the number of window groups of a given window group priority running in all window server sessions.

This function is the same as `WindowGroupList()` described above, but allows the application to restrict the list of window groups to those of a particular window group priority.

Parameters

`TInt aPriority`	Window group priority
`CArrayFixFlat<TInt> *aWindowList`	List of identifiers of all window groups in server of priority `aPriority`

Return value

`TInt`	`KErrNone` if successful, otherwise another of the system-wide error codes.

Enumerations

Enum TComputeMode

```
TComputeMode
```

Description

Compute mode flags.

When a window group takes focus or loses it, the window server can boost its client's thread or process priority to provide a better response to the user. How it alters the priority is determined by the current compute mode of the client.

See also:

- ComputeMode()

EPriorityControlDisabled	Client priority is permanently set to its current level — it is not altered or set by the windowing system if the focus changes. Thus if ComputeMode() is called with this flag when a client is in the foreground, it will subsequently have foreground priority even if it is in the background.
EPriorityControlComputeOn	Client process's priority is always set to EPriorityBackground.
EPriorityControlComputeOff	Client process's priority is set to EPriorityForeground when the window group takes focus, and set to EPriorityBackground when it loses focus. This is the default behaviour.

Enum TLoggingCommand

TLoggingCommand

Support

Supported from 7.0

Description

Window server logging commands passed to LogCommand().

See also:

- RWsSession::LogCommand()

ELoggingEnable	Enables logging.
ELoggingDisable	Disables logging.
ELoggingStatusDump	Logs the current status of all the windows in the tree, even if logging is not currently enabled.
ELoggingHeapDump	Logs information about the window server's heap size and usage, even if logging is not currently enabled.